EECS C106B: Project 2 - Nonholonmic Control *

Due date: March 8th at 11:59pm

Path Planning for the Bicycle Model

You will develop path planners for a second order nonholonomic system (e.g. cars) using three methods. You'll use an optimization-based planner, a modified RRT, and a sinusoidal steering method based off the constructive controllability material you've been learning in class.

Contents

1 Theory **2**
- 1.1 Path Planning for Nonholonomic Systems . . . 3
- 1.2 Optimization-Based Planning . . . 3
- 1.3 The Representation . . . 3
  - 1.3.1 The Dynamics . . . 3
  - 1.3.2 The Cost Function . . . 4
- 1.4 The Obstacles . . . 4
- 1.5 RRTs for Nonholonomic Systems . . . 5
  - 1.5.1 The Distance Function: Metrics on SE(2) . . . 6
  - 1.5.2 The Local Planner . . . 6
  - 1.5.3 Sampling . . . 6
- 1.6 Bang-Bang Control . . . 7
- 1.7 Steering with Sinusoids . . . 7
  - 1.7.1 The Algorithm . . . 8
  - 1.7.2 Binary Search for Determining Amplitudes . . . 9
  - 1.7.3 Dealing with Constraints . . . 9
- 1.8 Tracking Nonholonomic Open Loop Trajectories (Grad Students) . . . 9

2 Project Tasks **10**

3 Deliverables **10**

4 Getting Started **11**
- 4.1 GitHub Classroom . . . 12
- 4.2 Pulling starter code updates . . . 12
- 4.3 Environment Setup . . . 12
- 4.4 Running Planners . . . 12
- 4.5 Starter Code . . . 13
- 4.6 Implementation Tips . . . 14
- 4.7 Using the TurtleBots . . . 14
- 4.8 Common Issues . . . 15

5 Scoring **15**

6 Submission **15**

7 Improvements **16**

8 References **16**

*Developed by Valmik Prabhu and Chris Correa, Spring 2019. Further expanded and developed by Valmik Prabhu, Amay Saxena, and Tiffany Cappellari, Spring 2020, and again by Valmik Prabhu and Amay Saxena in Spring 2021.

1 Theory

For this project, you'll be running a script that makes a unicycle model robot behave like a bicycle model.

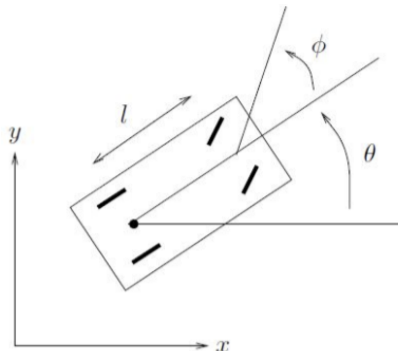

Figure 1: Dynamics of the Bicycle Model

As seen in the homework, the bicycle model has the following dynamics [1]:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \\ \frac{1}{l}\tan(\phi) \\ 0 \end{bmatrix} u_1 + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} u_2 \tag{1}$$

where u_1 is the car velocity, and u_2 the steering rate. Additionally, we will also have some constraints on our state variables. For instance, there will be constraints on (x, y) based on how large the environment is, and a constraint on the steering angle ϕ. There will also be lower and upper bounds on the control inputs (u_1, u_2). Much of this project will involve finding ways of dealing with these constraints while coming up with a feasible motion plan for the car.

As you can see the two input vector fields g_1 and g_2, where

$$g_1 = \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \\ \frac{1}{l}\tan(\phi) \\ 0 \end{bmatrix} \tag{2}$$

and

$$g_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{3}$$

do not span the whole state space $\mathbb{R}^4$. In order to examine the steerability of the system, we examine the Lie algebra, which is found by taking successive Lie Brackets of the input vector fields. The Lie bracket represents the second order terms that arise from alternating forward and reverse control of two inputs, and can be calculated through

$$g_3 = [g_1, g_2] = \frac{\partial g_2}{\partial x} g_1 - \frac{\partial g_1}{\partial x} g_2 \tag{4}$$

$$g_4 = [g_1, g_3] \tag{5}$$

$$g_5 = [g_2, g_3] \tag{6}$$

and so on. See homework 3 for more information. Working out the Lie Brackets shows that g_3 controls θ, and g_4 controls x and y perpendicular to the direction the car is facing. Both of these directions are constrained by velocity constraints, but by switching between steering left and right and driving forwards and backwards, you can generate motion in these directions.

1.1 Path Planning for Nonholonomic Systems

In this project, you'll be implementing a couple different path planners for this nonholonomic system. Note however that path planning for nonholonomic systems is a very open research problem, and that all of the methods presented here are imperfect in many ways. You'll be developing, in no particular order:

- An optimization-based planner. You'll be taking the path planning problem, discretizing it, and converting it into a nonlinear programming problem, which will be fed to an optimizer. You'll be implementing this part of the project in Python as part of your homework. You will then place the file you implemented as part of your homework into the `planners/` folder of the starter code. This should be all you need to code for this part of the project.
- A modified RRT designed to work for second order nonholonomic systems. By using clever motion primitives and sampling heuristics, we can get (reasonably) fast planning.
- A sinusoidal path planner based on this paper by Sastry and Murray [2]. This planner constructively generates a feasible path in essentially finite time, but it has some issues with obstacles.

1.2 Optimization-Based Planning

One way to solve the path planning problem is to format it as a nonlinear optimization problem. This method allows us to get an "optimal" path according to a chosen metric, and incorporate any number of constraints, from obstacles to input bounds to nonlinear and/or nonholonomic dynamics. Unfortunately, this method's primary technique is also its major disadvantage: Nonlinear optimization problems are usually nonconvex. This means that it's often impossible to prove the existence of a solution and that any solution we find is only guaranteed to be a local optimum, rather than the global optimal solution. There are many ways to model a path planning optimization problem, but the one we'll be using is

$$
\begin{aligned}
q^*, u^* = \operatorname{argmin} & \sum_{i=1}^{N} \left((q_i - q_{goal})^T Q (q_i - q_{goal}) + u_i^T R u_i\right) + (q_{N+1} - q_{goal})^T P (q_{N+1} - q_{goal}) \\
s.t.\ & q_i \geq q_{min},\ \forall i \\
& q_i \leq q_{max},\ \forall i \\
& u_i \geq u_{min},\ \forall i \\
& u_i \leq u_{max},\ \forall i \\
& q_{i+1} = F(q_i, u_i),\ \forall i \leq N \\
& (x_i - \text{obs}_{j_x})^2 + (y_i - \text{obs}_{j_y})^2 \geq \text{obs}_{j_r}^2,\ \forall i, j \\
& q_1 = q_{start} \\
& q_{N+1} = q_{goal}
\end{aligned}
$$

1.3 The Representation

A path at its core is a function $q(t)$ that describes the system state over time, and possibly a function $u(t)$ which describes the open-loop control inputs needed to drive the system along this path [3], but there are many ways to represent such a path. By far the most common parameterization is via waypoints, or an array of states and inputs over time $[q, u, t]$. These waypoints may either be at constant intervals [3, 4] or not [5]. If waypoints are at constant intervals, you'll need to choose a time T, or a number of waypoints N in which the system must reach its goal. Generally, using nonconstant time intervals requires you to add the times t_i for each waypoint as variables to be optimized alongside q_i and u_i. This makes the optimization problem much harder, but allows the system to reach the goal in variable time. We'll be using waypoints with constant time intervals in this planner. Other ways to represent paths are via splines [6, 7] or polynomials, or via more esoteric methods like Riemannian metrics [8, 9].

1.3.1 The Dynamics

In order to work with waypoints, which are a *discrete* representation of a path, we need our dynamics to be discrete as well. The ideal way to discretize our system would be to integrate our states over the time between inputs:

$$q_{N+1} = \int_{t_N}^{t_{N+1}} F(q_N, u_N) dt$$

However, integrals can be rather computationally expensive to compute. Since we'll be solving our dynamics many times during the optimization (the example in your homework requires at least 10,000 computations, and likely many more), we'll have to use some computationally feasible approximations. The easiest approximation of an integral is an Euler integral:

$$x(t + \delta t) = \int_{t}^{t+\delta t} x(t)dt \approx x(t) + \dot{x}(t)\delta t$$

This approximation actually stems from the definition of a derivative

$$\dot{x} = \lim_{\delta t \to \infty} \frac{x(t + \delta t) - x(t)}{\delta t}$$

So as δt approaches zero, the Euler approximation exactly equals the value of the integral. Since we're using waypoints separated by constant time intervals, the discretized dynamics will be the same between every pair of waypoints.

The Euler integral is a first order approximation of an integral. We could use higher order derivatives to make it more accurate. This is called a Runge-Kutta approximation. We could also use collocation methods to better approximate the integrals [5]. In general, more accurate integration methods are more computationally expensive, but the additional accuracy becomes necessary for highly dynamic systems like quadcopters or legged robots.

1.3.2 The Cost Function

The cost function is the optimality metric by which we judge our path. We choose to use a very simple cost function:

$$J = \sum_{i=1}^{N} \left((q_i - q_{goal})^T Q (q_i - q_{goal}) + u_i^T R u_i \right) + (q_{N+1} - q_{goal})^T P (q_{N+1} - q_{goal})$$

This cost function [1] has two components: the *stage cost* $(q_i - q_{goal})^T Q (q_i - q_{goal}) + u_i^T R u_i$ is the cost at each timestep i. Q penalizes the distance from the goal of the system at every timestep, while R penalizes the input at each timestep. $(q_{N+1} - q_{goal})^T P (q_{N+1} - q_{goal})$ is called the *terminal cost*, and it penalizes the distance from the goal of the last timestep. This cost isn't actually necessary in our case, since we have a constraint that the last state be the goal. However, sometimes this constraint isn't possible to fulfill (we haven't given the system enough time, the state isn't reachable, or our constraint tolerance is too tight). In this case, we can remove that constraint and rely on our P cost to get us as close as possible. Usually P is set as Q or some scalar multiple of it.

The cost function can potentially evaluate much more than distance and energy (input) usage. Often smoothness (minimization of $q_i - q_{i-1}$ or $u_i - u_{i-1}$), safety (distance from obstacles) or other "soft" constraints will be added to the cost function as well [3].

1.4 The Obstacles

Representing obstacles is often one of the hardest aspects of path planning. There are two main problems in obstacle representation. The first is how to represent arbitrary obstacles as collections of primitives, and the second is how to represent these primitives in the robot's configuration space.

For the first problem, there are a couple ways to represent obstacles. The easiest is to use circles or spheres (depending on if you're planning in 2D or 3D space). It's easy to compute the distance to a sphere, but unfortunately using spheres result in quadratic constraints, which will often slow down your optimizer. Obstacles in the real world are usually not spheres however, so often planners will represent arbitrary obstacles as collections of spheres, with some extra buffer space to avoid accidental collision. However, this can potentially restrict the free space enough to make the problem infeasible (this can occur in so-called "narrow passages", which are a common research topic in path planning).

More complex object primitives, like ellipses or polytopes (polygons in n dimensional space) can be used. Polytopes, which are just collections of hyperplanes, can potentially decrease computation time, but often more complex primitives increase computation time [4, 10]. You can also use signed distance fields [3], which are also called costmaps. These maps can be useful in that they can represent arbitrarily shaped obstacles, and provide gradient information no matter the shape, but are problematic in that they are stored as discrete arrays and therefore require

[1] Note: this cost function looks very similar to the cost function used in LQR control. This is no accident. We're actually solving the nonlinear version of the constrained finite time discrete LQR problem [10].

the space to be gridded. As a result, they're only suitable in spaces of lower dimension (like 3D space, for example).

The other problem in representation is representing the obstacles, which are normally defined in the real world, in the configuration space of the robot. Even for simple robots, the representation of the obstacle in the configuration space can be somewhat difficult to compute, as can be seen in figure 2. For more complex robots like arms, it's impossible to represent obstacles in the configuration space (why?), so researchers need to use clever tricks to perform optimization [3, 4].

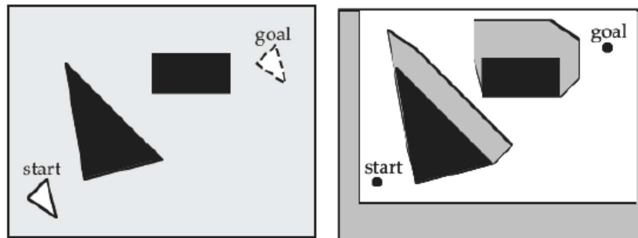

Figure 2: Simple workspace obstacles represented in the configuration space of a triangular robot [11]. How do you think the representation would change if the robot could spin?

The final problem is how to ensure safety between waypoints. If you're using waypoints with constant intervals, you can calculate the maximum difference $\Delta q = q_{i+1} - q_i$ and enlarge your obstacles enough that a path of that length cannot collide with an obstacle. If δt is small this is a reasonable assumption to make, but if it's variable or large you'll need to account for these collisions [3, 4].

1.5 RRTs for Nonholonomic Systems

RRT (which stands for rapidly-exploring random tree) is a sampling based motion planner, and is one of the most commonly used motion planning algorithms in industry [12]. RRT works by iteratively building a graph on the configuration space, starting from a start configuration, until we reach within some target neighborhood of the goal configuration.

The standard RRT algorithm proceeds as follows. The algorithm takes as input a start configuration s, a goal configuration g, a goal threshold ϵ, and a configuration space C which has knowledge of the environment (such as obstacles or the robot model).

1. Initialize an empty graph G. The graph will have robot configurations as vertices. Edges between configurations will be motion plans that take the robot from the first configuration to the next.
2. Add the start configuration s to G.
3. Until we either succeed or exhaust the maximum allowable number of iterations, do:
   (a) Sample a configuration c_{rand} from C at random. If this configuration is in collision with an obstacle, discard it and retry.
   (b) Find the nearest node in G to c_{rand}. Call this node c_{near}.
   (c) Construct a local plan p that steers the robot from c_{near} to c_{rand}. If this plan is in collision with an obstacle, discard it and return to (a).
   (d) Take some small prefix of this plan. i.e. instead of following the whole plan and going all the way to c_{rand}, construct a new plan p' that is simply p followed for a small Δt amount of time. The rationale here is that we are taking a small step in the direction of c_{rand}. This path p' ends at some configuration c_{new}. Add this configuration as a vertex to G. Add the path p' as an edge between c_{near} and c_{new} to G.
   (e) If c_{new} is within ϵ of the goal g, then we have success. Treat c_{new} as your final configuration. Reconstruct the path from the start to c_{new}, and return it as our motion plan.

When planning for the bicycle model, there are a few key ways in which the above algorithm needs to be modified. First off, our configuration space C is the state space of a robot, and is the set of all (x, y, θ, ϕ) states that the robot can be in. Our start and goal configurations are given in this state space. So, the first thing we need to decide on is how we measure distances in this space. It is not sufficient to just consider the (x, y) distance between two

configurations, since we also care about aligning the heading angle θ. So we need to come up with a distance function that considers at least the x, y, θ variables. This will allow us to plan for the heading angle as well, and will allow us to perform manipulability tasks like parallel parking and point turns.

The second consideration concerns the local planning step. Step 3 (c) involves constructing a local motion plan that can take the robot from the configuration c_{new} to the configuration c_{rand}. However, in the case of the bicycle model, this becomes tricky since we need to find a local plan that respects the dynamics of the robot. The trick here will be to take advantage of the fact that in step 3 (d) we decide to move only some prefix of length Δt along the local plan. So, instead of finding a complete motion plan to the sampled vertex, we will instead directly choose a local plan from a set of possible plans of length Δt. These small motions will be called *motion primitives*, and at each step we will simply pick the motion primitive that brings us closest to the sampled configuration. This will allow us to quickly perform the local planning step, since we can pre-determine the set of motion primitives we will be choosing from.

The next two sections describe these two considerations in a bit more detail.

1.5.1 The Distance Function: Metrics on SE(2)

The distance function is an important design decision, and will affect how well your algorithm performs. The first observation we can make is that we only need to consider the states (x, y, θ) in our distance function. It suffices to worry only about these variables, since we have direct control over ϕ so we can set it to its desired value once we have aligned the other variables. With these three variables, we have a parameterization of $SE(2)$, and we can use any metrics that work on $SE(2)$.

A naive approach might be to simply take the cartesian distance in (x, y, θ) space. This is a bad idea, however, since then the angles 0 and 2π look like they are very far away, when in fact they are the same. So we need to pick metrics that respect the periodic nature of the θ state variable. You can find a number of good metrics for motion planning in $SE(2)$ from various motion planning textbooks. The relevant section of *Planning Algorithms* by Steven M. LaValle is here [13]. The rest of the textbook is also a good read and has many relevant sections related to sampling based motion planning.

1.5.2 The Local Planner

As mentioned above, one way to design our local planner is to simply have it pick between a predetermined set of motion primitives. These should be a set of motions that sufficiently span the set of movements available to our bicycle model robot. Remember that an RRT will simply return a path constructed by chaining together edges in the graph. If each edge is going to be one of our motion primitives, we need to be careful to pick a set of motion primitives that are expressive enough that we can move from any configuration to any other configuration by chaining them together (otherwise we have no hope of finding a solution).

For instance, one set of motion primitives might be {moving forward, moving backward, moving forward while turning left, backward while turning left, forward while turning right, backward while turning right}. This wouldn't be sufficient, however, as you also want to allow the planner to pick from a variety of different speeds. You should spend some time designing a good set of motion primitives. Another way to pick motion primitives is to perhaps start off with some nice sequence of control inputs, and see what kinds of paths they generate (for instance, what does the robot do if we command it to move forward at constant velocity a_1, and steer according to a sinusoid $\dot{\phi} = a_2 \cos(\omega t)$?). You may also want to look at Dubins/Reeds-Shepp Paths for inspiration [14, 15].

1.5.3 Sampling

A final consideration is the sampling strategy. You may find that simply sampling uniformly at random does not lead to very good performance. Instead, you may find that you need to implement better *sampling heuristics*. For instance, you may want to bias your sampler toward making progress towards the goal. A couple possible sampling heuristics to this end are:

- *Goal-bias:* Pick a number $p \in (0, 1)$. With probability p, sample the goal directly. Else, sample at random.
- *Goal-zoom:* Pick a number $p \in (0, 1)$. With probability p, sample from a ball of radius $\min_i d(x_i, g)$ centered at the goal g, where $x_1, ..., x_n$ are the configurations currently in your graph (and d is your distance function).

Feel free to design other heuristics that you find work well.

1.6 Bang-Bang Control

In class, you initially learned that you could steer your system in nonholonomically constrained directions by alternating your control inputs. This is known as bang-bang control. While bang-bang control can work well as a control strategy in other situations, actually actuating a Lie bracket using bang bang control works pretty badly. You end up getting a vibrating, jerky trajectory that drifts unpredictably if the time between alternations isn't differentially small. This is why we use sinusoidal steering. We've provided a file called `bang_bang.py` that you can play around with to gain some intuition about Lie brackets.

1.7 Steering with Sinusoids

You'll also be implementing the steering with sinusoids algorithm from this paper by Sastry and Murray[2]. They prove that a good way to control nonholonomic systems is, rather than using a bang-bang "square-wave" approach, to control the inputs with out-of-phase sinusoids. To do this, they require that the system dynamics approximate a canonical model

$$\begin{aligned}\dot{x}_1 &= u_1\\ \dot{x}_2 &= u_2\\ \dot{x}_3 &= x_2u_1\\ \dot{x}_4 &= x_3u_1\end{aligned}$$

Our car model does not match this canonical form, but can be made to approximate it by applying a nonlinear transformation

Canonical Car Model

$$\begin{aligned}\dot{x} &= v_1, & v_1 &= \cos(\theta)u_1\\ \dot{\phi} &= v_2, & v_2 &= u_2\\ \dot{\alpha} &= \frac{1}{l}\tan(\phi)v_1, & \alpha &= \sin(\theta)\\ \dot{y} &= \frac{\alpha}{\sqrt{1-\alpha^2}}v_1\end{aligned}$$

As you can see, this model approximately matches the canonical form, with

$$\begin{aligned}\dot{x}_1 &= u_1\\ \dot{x}_2 &= u_2\\ \dot{x}_3 &= f(x_2)u_1\\ \dot{x}_4 &= g(x_3)u_1\end{aligned}$$

Even though $\dot{x}_3$ and $\dot{x}_4$ have nonlinear terms, we can use Fourier analysis to get rid of them. Another problem is not so easy to get rid of; this model contains a singularity at $\theta \in \{90°, -90°\}$. When the robot is turned 90°, we know that all the input velocity u_1 goes in the y direction, and everything works normally. However, when that occurs in this model, u_1 goes to infinity, $\dot{\alpha}$ goes to zero, and and $\dot{y}$ goes to $\frac{0}{0}$. One potential solution is to match the canonical model in a different way, by replacing x with y as the first canonical state. The model would look something like this:

Alternate Model

$$\begin{aligned}\dot{y} &= v_1, & v_1 &= \sin(\theta)u_1\\ \dot{\phi} &= v_2, & v_2 &= u_2\\ \dot{\alpha} &= -\frac{1}{l}\tan(\phi)v_1, & \alpha &= \cos(\theta)\\ \dot{x} &= \frac{\alpha}{\sqrt{1-\alpha^2}}v_1\end{aligned}$$

This is also an approximation of the canonical system, except this system is undefined at $\theta \in \{0°, 180°\}$. You could simply switch models depending on where θ happens to be, perhaps even within one plan. Another thing you could do is ignore the canonical form and use $\dot{\theta} = \frac{1}{l}\tan(\phi)u_1$ for planning changes to θ (but not y), because as you'll see, prescribing a direction for x is unnecessary in computing an open loop control law for θ.

1.7.1 The Algorithm

The algorithm for developing an open-loop path for steering is deceptively simple, but contains a couple very tricky bits. It's conducted in three stages:

1. First, steer the x_i's, which in this case are x and ϕ to their desired states. This is reasonably simple, and requires choosing $v_i = (x_{i_d} - x_i)/\Delta t_d$, where Δt_d is the desired time to complete the maneuver.

2. Second, steer x_{ij}, which in this case is θ or α to its desired state. To do this, set

$$v_1 = a_1 \sin(\omega t)$$
$$v_2 = a_2 \cos(\omega t)$$

The maneuver will run for $\frac{2\pi}{\omega}$ seconds (one period), and its magnitude will be determined by a_1 and a_2. The tricky bit comes when trying to select a_1 and a_2 to move some desired magnitude in θ. If the system followed the standard canonical form, the difference would simply be $x_{ij}(\frac{2\pi}{\omega}) = x_{ij}(0) - \frac{a_1 a_2}{\omega}$. However, since our model is nonlinear, this doesn't work. To find the change in α, we first look at the change in ϕ. Since $\dot{\phi} = a_2 \cos(\omega t)$ we know

$$\phi(t) = \phi(0) + \frac{a_2}{\omega} \sin(\omega t) \tag{7}$$

Note that $\phi(\frac{2\pi}{\omega}) = \phi_0$, which demonstrates that these sinusoids don't change x_i, only x_{ij}. Thus,

$$\dot{\alpha}(t) = f(\phi(t))v_1(t) = f\left(\frac{a_2}{\omega} \sin(\omega t) + \phi(0)\right) a_1 \sin(\omega t) \tag{8}$$

We could find $\alpha(\frac{2\pi}{\omega})$ by integrating, but this integral is very difficult to compute. One way to (slightly) simplify it is to use a Fourier expansion.

As you likely learned in your lower division math classes, periodic functions can be represented as a summation of sinusoids called the Fourier Series. By taking this expansion, we approximate

$$f\left(\frac{a_2}{\omega} \sin(\omega t)\right) = \beta_1 \sin(\omega t) + \gamma_1 \cos(\omega t) + \beta_2 \sin(2\omega t) + \gamma_2 \cos(2\omega t) \cdots \tag{9}$$

Which means that our integral becomes

$$\alpha(\frac{2\pi}{\omega}) = \alpha(0) + \int_0^{\frac{2\pi}{\omega}} \beta_1 \sin(\omega t) a_1 \sin(\omega t) + \gamma_1 \cos(\omega t) a_1 \sin(\omega t) + \beta_2 \sin(2\omega t) a_1 \sin(\omega t) + \cdots dt \tag{10}$$

Now, we use the property that sinusoids of integrally related frequencies are mutually orthogonal over their period. Two functions $f(x)$ and $g(x)$ are orthogonal on $x \in [a, b]$ if $\int_a^b f(x)g(x)dx = 0$. The fact that the terms of the Fourier sine series are orthogonal make sense since the sinusoids essentially form a basis with weights β and γ. By using this property, all but the first term in the integral disappears, leaving us with

$$\alpha(\frac{2\pi}{\omega}) = \alpha(0) + \int_0^{\frac{2\pi}{\omega}} \beta_1 a_1 \sin(\omega t)^2 dt = \alpha(0) + \frac{\pi a_1 \beta_1}{\omega} \tag{11}$$

This is nice and easy. Of course, β_1 still requires an integral to compute:

$$\beta_1 = \frac{\omega}{\pi} \int_0^{\frac{2\pi}{\omega}} f\left(\frac{a_2}{\omega} \sin(\omega t) + \phi(0)\right) \sin(\omega t) dt \tag{12}$$

You'll have to compute this integral numerically and find an a_1 and a_2 that result in a desired translation of x_{ij}. You may have to implement binary search for this, or some other method to find these values. Also note that it may be easier to numerically integrate α directly, rather than numerically finding β_1. However, this may make it much more computationally intensive to find a_1 and a_2.

3. The third step is to use integrally related sinusoids to control x_{ijk}, in this case y, without changing any of the other variables. Here, we choose

$$v_1 = a_1 \sin(\omega t)$$
$$v_2 = a_2 \cos(2\omega t)$$

Once again, we have to deal with a very hairy integral in order to figure out $y(\frac{2\pi}{\omega})$. We want to find

$$y(\frac{2\pi}{\omega}) = y(0) + \int_0^{2\pi} \omega g(\alpha(t)) a_1 \sin(\omega t) dt \tag{13}$$

Where

$$\alpha(t) = \alpha(0) + \int_0^t f(\frac{a_2}{2\omega}\sin(2\omega\tau) + \phi(0)) a_1 \sin(\omega\tau) d\tau \tag{14}$$

We use the Fourier Series to get

$$y(\frac{2\pi}{\omega}) = y(0) + \int_0^{2\pi} \omega\beta_1 \sin(\omega t) a_1 \sin(\omega t) + \gamma_1 \cos(\omega t) a_1 \sin(\omega t) + \beta_2 \sin(2\omega t) a_1 \sin(\omega t) + \cdots dt \tag{15}$$

Which becomes

$$y(\frac{2\pi}{\omega}) = y(0) + \frac{\pi a_1 \beta_1}{\omega} \tag{16}$$

where

$$\beta_1 = \frac{\omega}{\pi}\int_0^{\frac{2\pi}{\omega}} g\left(\alpha(t)\right)\sin(\omega t) dt = \frac{\omega}{\pi}\int_0^{\frac{2\pi}{\omega}} g\left(\int_0^t f(\frac{a_2}{2\omega}\sin(2\omega\tau) + \phi(0)) a_1 \sin(\omega\tau) d\tau + \alpha(0)\right)\sin(\omega t) dt \tag{17}$$

You can expand this out, but it won't be very pretty. You'll still need to do quite a lot of numerical integration. In fact, you'll assuredly have to do binary search here, since β_1 depends on a_1 as well as a_2. Previously, you could set an arbitrary a_2, find β_1, and then set a_1 to produce the desired displacement. Now you'll need to do binary search on a_1 to find the correct combination of a_1 and β_1 in order to generate the desired displacement.

While finding a_1 and a_2 for general constraints is particularly difficult, there are ways to simplify the model you use for certain conditions. Furthermore, I suggest that you use a heuristic method to get things somewhat working before implementing all the integrals.

1.7.2 Binary Search for Determining Amplitudes

When controlling y, you want to find some values a_1 and β_1 such that

$$\frac{\pi a_1 \beta_1}{\omega} = y\left(\frac{2\pi}{\omega}\right) - y(0) \tag{18}$$

Unfortunately, β_1 depends on a_1, so you can't just set β_1 to an arbitrary number (by setting a_2 to an arbitrary number) and find an a_1 that will work. You have to set a_2 and then find a_1 *dynamically.* Luckily, β_1 happens to be *monotonic* as a function of a_1; if the steering amplitude is small enough (within reason), then the faster (and farther) you drive forward, the more you'll travel in y. This means that given an a_2 you can use binary search to efficiently find the best a_1.

1.7.3 Dealing with Constraints

The steering with sinusoids algorithm was originally designed to work with unconstrained systems, but can reasonably be modified to work with state and input constraints by using multiple motions on each state rather than a single motion. State constraints on x and ϕ can be easily implemented as limits on the amplitudes of your sinusoidal inputs. You won't need to deal with limits on θ/α, but how would you deal with limits on y?
Input bounds on $\dot{\phi}$ are similarly easy to implement, but bounds on v will be harder to implement. You may need to use heuristics to properly deal with them, and it's fine to be a bit more conservative on your constraint checking than you absolutely need to.

1.8 Tracking Nonholonomic Open Loop Trajectories (Grad Students)

Creating these open-loop trajectories is all well and good, but as you've seen, executing trajectories in the real world never works perfectly. Thus, in order to properly execute trajectories provided by the above planner, we need to add feedback control. Tracking trajectories for nonholonomic systems is also a tricky problem, but for common systems (like the unicycle or the bicycle model) many controllers already exist. Thus, we've decided to make this section very open-ended. Grad students (or extra credit for undergrads) should implement a control law that will drive tracking error to zero. You may use any control technique you'd like, be it from a paper or your own creativity. Feel free to use any heuristics you'd like as long as you can justify their use.

- Remember that heuristic techniques like cascaded control, point-offset methods, and gain scheduling can be powerful tools. Feel free to use PID as well (though the more work you want your PID to do the harder it'll be to tune).
- There are a number of the papers that cover tracking control for mobile robots [16–19].
- Lots of controllers [6, 20] exist for the unicycle model. In many cases you can adapt them to the bicycle model.
- Professor Sastry has a number of papers on tracking control for nonholonomic systems such as [21]. Feel free to try implementing some of these more rigorous papers, but they can be challenging to work with.

2 Project Tasks

For this project, you'll write three types of planners to plan nonholonomic motions. In the starter code, you have been given a number of environments that you can load onto the simulator and test your planners. These are `empty.yaml`, an empty 5m x 5m space, `map1.yaml`, a 10m x 10m space with 2 obstacles, and `map2.yaml`, a 10m x 10m space with 5 obstacles. See the Getting Started section for instructions on how to use these yaml files to set up an environment. You should use your planners to execute the following tasks.

1. **Manipulation:** Using the sinusoid, optimization, and RRT planners, execute the following trajectories in the **empty** environment:
   - Plan from position $(1, 1, 0, 0)$ to $(2, 1.3, 0.7, 0)$. (simple motion)
   - Plan from position $(1, 1, 0, 0)$ to $(1, 3, 0, 0)$ (parallel park).
   - Plan from position $(1, 1, 0, 0)$ to $(1, 1, \pi, 0)$ (point turn).

   Watch out! The robot has state and input bounds so you'll have to be careful with how you design your sinusoidal planner.

2. **Navigation:** Using the RRT and optimization based planners, execute the following tasks:
   - In **Map 1** - navigate from $(1, 1, 0, 0)$ to $(9, 9, 0, 0)$.
   - In **Map 2** - navigate from $(1, 1, 0, 0)$ to $(9, 9, 0, 0)$.

3. Execute the trajectories you generated above for both the manipulation task and navigation task on the simulator. Discuss how well the simulator is able to track your trajectories with open loop inputs. Does the simulator have an easier time accurately following the trajectories generated by some planners more than others? Why or why not?

4. Execute the three manipulation trajectories from all three planners on a TurtleBot. It will likely fail (miserably). Why do you think it doesn't work very well?

5. (**Grad Students**) Design a closed loop controller for bicycle model plans, and tune it until it works on all the paths in simulation. Once you've tuned to satisfaction, try running the controller for the three manipulation trajectories from all three planners on a TurtleBot. It will still likely fail. Why do you think this is?

3 Deliverables

To demonstrate that your implementation works, deliver the following in a report. The purpose of these project reports are to gradually scale up to a full conference-style paper, which you'll be writing for your final project. Please format your report using the IEEE two-column conference template. Column suggestions do not account for the figures.

1. Abstract: An abstract, of at most 150 words, describing what you did and what results you achieved. Conveying information concisely is a difficult skill, and we want you to practice it here.

2. Methods: Explain in detail your approach towards each of the three planners, making sure to address the following questions. You can expect the readers of your report to know everything in this manual, so you don't need to repeat any of it (though you should reproduce equations and the like as needed for readability).

(a) For the sinusoidal planner, how did you account for input constraints, state bounds, and the singularity? (∼ 2 columns)

(b) For the optimization-based planner, how did you decide what to set for N and δt? How might you do this for an arbitrary goal state? (< 1 column)

(c) For the RRT planner, how did you design your local planner? What distance metric did you use, and what (if any) sampling heuristics did you use? If appropriate, provide a figure illustrating the motion primitives you used. (∼ 2 columns)

3. Experimental Results: Summarize the performance of each of the planners on each scenario, using both words and figures. (< 1 column, plus figures)

   (a) For each manipulation task, provide plots of the returned path for all three planners.

   (b) For each navigation task, provide plots of the returned path for the RRT and optimization-based planners.

   (c) Remember that each figure should come with a detailed caption. An informed reader should be able to understand the figure without having to read the paper at all.

4. Discussion: (∼ 1-2 columns)

   (a) Discuss the performance of each planner. What worked, what didn't, and why?

   (b) Compare and contrast the performance of the planners against one another. When would you use each type of planner? How might you improve them?

   (c) How did your open loop plans fare in simulation? Why do you think any failures occurred?

   (d) How did your open loop plans fare on the TurtleBot? Why do you think any failures occurred?

   (e) Why doesn't the sinusoidal planner work for navigation tasks? How might you modify the steering with sinusoids algorithm to work with obstacles?

   (f) **Grad Students:** Describe your control law in detail along with whatever math you've used to justify it. Discuss its ability to track paths in simulation. How does it fare tracking paths produced by each planner? What about paths on each task? How does it fare on the real robot? We expect you to illustrate your results with figures and/or data of your choice.

5. Bibliography: A bibliography section citing any resources you used. This should include any resources you used from outside this class. Please use the IEEE citation format.

6. Appendix:

   (a) GitHub Link: Provide the link to your GitHub classroom repo. Simply push your code to the private repository that was created for your team, and we will be able to see any changes you push to your assignment repository.

   (b) A video screencap of: your sinusoidal planner working in simulation with all three manipulation tasks, and your other planners running in simulation in one of the navigation tasks. Additionally, include videos of all three planners executing the parallel park and point turn trajectories on the TurtleBot. All these videos should be stitched into one video.

   (c) Bonus: What do you think the learning goals of this assignment are? How effective was this assignment at fostering those learning goals for you? How can the lab documentation and starter code be improved? We are especially looking for any comments you have regarding the pedagogical success of the assignment.

7. Page Limit: To prepare you for conference-style papers (and to protect the graders' time), reports are **limited to 6 pages**, not including the bibliography or appendix.

   (a) While we are not enforcing a figure limit, we expect your figures to be concise, informative, and readable, with detailed captions for each. We expect you to need 6-8 figures in total.

4 Getting Started

Create a new ROS workspace to store your code for this project. Remember to build and source your workspace after you download the starter code.

4.1 GitHub Classroom

For projects in this class, we will be using GitHub Classroom. To access the starter code, simply head over to the Project 2 assignment page to accept the assignment. If you are asked to associate yourself with a school identifier, just click "Skip to the next step".

You should now be asked to create a team or join from a list of existing teams. If one of your teammates has already created a team, you should join that team instead of creating a new one. **After you have joined a team, you will not be able to switch teams by yourself. If you make a mistake or something else comes up that requires you to switch teams, let us know.**

Your team should now have a private project repository located at https://github.com/ucb-ee106-classrooms/project-2-your-team-name. Let us know if anything went wrong.

Once you have formed your team, fill out this form so we can assign your team to a TurtleBot. Note that you can work on most of this project without a TurtleBot, and reserving any robot (both the TurtleBots and the arm robots) will also reserve the computer that the robot is associated with.

4.2 Pulling starter code updates

If there are any updates to the starter code that you wish to pull you may do so with the commands

```
cd your-repo
git remote add public https://github.com/ucb-ee106/proj2_pkg.git
git pull public main
git push origin main
```

4.3 Environment Setup

Recall from homework 3 that we will be using CasADi as our optimization back-end. If you don't already have it installed, use

```
pip install casadi
```

For this project we will be using STDR Simulator to simulate our robot and test our code. This package allows us to simulate both the robot and various obstacles (the latter being a feature that Turtlesim unfortunately lacks and why we are using this instead). First we need to clone the package.

```
git clone https://github.com/stdr-simulator-ros-pkg/stdr_simulator.git
```

To create a map of obstacles STDR takes a `.png` file and a `.yaml` file and to simulate a robot it takes a `.yaml` file. We have created a simple robot for you to use already and a launch file that automatically spawns it onto your map. We have also provided you with a script to create new maps (`create_map.py`) as well as a few pre-generated maps that can be found in `proj2_pkg/maps/`. Feel free to check those out and create new maps (you can make new robots as well if you'd like but you don't need to for this project). To launch a simulation run

```
roslaunch proj2_pkg init_env.launch map:=empty.yaml start_position:="1 1 0"
```

replacing `empty.yaml` and `"1 1 0"` with whatever map and starting position you prefer. The default map is `empty.yaml` and the default start position is 1 1 0 (don't forget to put the start position inside quotes if you are changing it from the command line). Look inside the `maps` folder to see what maps we have provided for you or make your own using `create_map.py`. This also allows us to translate the default unicycle model to a bicycle model. You can now begin running your planners on the simulation!

4.4 Running Planners

Once you have `init_env.launch` running, you can use the script `main.py` to test your implementation. Look at the starter code section and read this file to figure out what arguments it takes.

This file will load information about the environment (such as obstacles, input limits and state limits, starting state of the robot, etc.) from the parameter server and initialize a `ConfigurationSpace` object using this information. It

will then pass this object into your planners to initialize it. The arguments to `main.py` allow you to specify the goal location of the robot along with what planner you would like to use. The script will then use your implementation to come up with a plan, visualize it, and then execute it.

4.5 Starter Code

- `src/proj2/converter/bicycle_converter.py` You should look through this file to make sure you know how it works, but hopefully you won't have to modify it at all. It subscribes to the `/bicycle/cmd_vel` topic which listens for `BicycleCommandMsg` messages. It publishes the bicycle state to the `/bicycle/state` topic as a `BicycleStateMsg` message. If you're confused at what is in these messages look at `proj2_pkg/msg`.
- `scripts/main.py` This is the script you will run to test your planner implementations. This script interfaces with the bicycle converter node and executes your generated plans using an open loop controller.

  You should edit the hyper-parameters passed into your planners from this file to tune them.

- `proj2_pkg/src/proj2/planners/sinusoid_planner.py` The first of the planner classes you will implement. You should implement the sinusoidal planner in this file. You should spend lots of time reading the steering with sinusoids paper. The high level goal of this class is to take in a goal pose, and then use the algorithm described in the paper to determine a trajectory in the form of a `Plan` object to steer the bicycle to the goal pose. The interface for the `Plan` data structure is defined in `configuration_space.py` (see below). You should create individual trajetories to steer x, ϕ, α, and y separately, then combine the trajectories together. Note that you'll need to deal with limits on x, y, ϕ, u_1 and u_2, and may therefore need multiple maneuvers to steer one state variable. We've provided you with basic implementations for `steer_x` and `steer_alpha`, though you may need to edit them depending on how you choose to account for state and input constraints.
- `proj2_pkg/src/proj2/planners/rrt_planner.py` This is an implementation of an RRT Planner. You do not necessarily need to edit this file, though you are free to do so. This class uses a `ConfigurationSpace` object to interface with the environment and robot model. The `ConfigurationSpace` object implements methods for checking collisions of points and plans, sampling new points in the configuration space, and creating local plans. You will implement these methods for a bicycle model robot in `configuration_space.py`. Although you may choose not to edit `rrt_planner.py`, you should still read through this file and familiarize yourself with how this file is interfacing with the configuration space.
- `proj2_pkg/src/proj2/planners/configuration_space.py` This file provides a template for the configuration space object which is used by your planners to interact with the environment. You should take a look at `rrt_planner.py` and `optimization_planner.py` to see how it is used. Each `ConfigurationSpace` class implements methods for checking collision of a given point or path with obstacles, sampling new configurations, and creating local plans, all of which are crucial parts of the RRT algorithm. Your job will be to implement the `BicycleConfigurationSpace` object, and fill in the function stubs in the file.
- `proj2_pkg/src/proj2/planners/optimization_planner.py` This class we have provided interfaces with your code from homework 3 and wraps it in the same interface as the other planners. You won't need to edit this file. You will simply place your completed `optimization_planner_casadi.py` file from homework 3 in the same folder as this file.
- `proj2_pkg/launch/init_env.launch` This launch file takes a `.yaml` and an initial position for the robot and starts an environment using the `.yaml` map. Don't forget to run this when you want your planner to run with obstacles in simulation. You don't have to edit this file but taking a look at what it's doing may be helpful. It also subscribes to the `/bicycle/cmd_vel` topic which listens for `BicycleCommandMsg` messages. It publishes the bicycle state to the `/bicycle/state` topic as a `BicycleStateMsg` message. If you're confused at what is in these messages look at `proj2_pkg/msg`.

  Essentially, this node has the job of abstracting away the robot so that it can accept bicycle model commands. Recall that the robots we are using are all unicycle model robots. This node offers us an interface wherein the current state of the robot in bicycle model coordinates is published to `/bicycle/state`. Additionally, it allows us to control the robot by publishing bicycle model commands to a topic `/bicycle/cmd_vel`. This node then handles converting those commands to equivalent unicycle model commands and sending them to the robot.

- `proj2_pkg/src/proj2/controller/controller.py` This currently implements an open loop velocity controller using a similar file structure to the starter code in Project 1.

- `proj2_pkg/maps/create_map.py` This file generates a `.png` image and a `.yaml` file to create maps for STDR Simulator. We have already provided you with a few maps to use for your project so you don't have to use this but feel free to use it to generate more maps for yourself to test things on.
- `scripts/bangbang.py` This is a script used to illustrate the "bang-bang" approach to controlling g_3 and g_4 in the Lie algebra. You don't need to turn it in or modify it at all (but feel free to do so). It is simply for your viewing pleasure.
- `scripts/sinusoids.py` This is a script used to illustrate the steering with sinusoids approach to controlling g_3 and g_4 in the Lie algebra. You don't need to turn it in or modify it at all (but feel free to do so). You should use this to check/debug the paths you produce in the sinusoid planner and examine what happens when you use various control inputs. There are two command functions in `sinusoids.py`, `cmd`, and `cmd_v`. The first actuates the normal inputs of the TurtleBot. The second performs the transformation needed to actuate the modified input for the steering with sinusoids formulation.

4.6 Implementation Tips

We expect this project to take approximately thirty hours per person for a group of three. Remember that you can work on all three planners simultaneously.

1. For the optimization planner, all you should need to do is place your completed file from homework 3 in the `planners/` folder, and then figure out if you need to change N and dt. If this part of the project takes you longer than 30-60 minutes, you're probably doing something wrong.
2. For the sinusoidal planner, you should only need to edit `sinusoid_planner.py`. The first thing you should do is fill in `steer_phi` and `steer_y`. Once you've filled in these functions you should be able to run the simple motion without any trouble. After this, you'll need to figure out how to overcome the state bounds and singularity. You may need to edit some or all of the functions provided, and you'll likely need to make more functions or change more things. This is one of the two main tasks in the project.
3. For the RRT planner, you only need to edit `configuration_space.py`. You'll need to fill in five functions in the `BicycleConfigurationSpace` class. While they should all be reasonably easy to implement, you should make sure that they run pretty fast so that your RRT doesn't take too much time. Expect to take a lot of time figuring out good primitives and sample heuristics, but remember that it doesn't have to be perfect. This is an open problem after all. This is the second main task in the project.
4. Based on your choice of metric, you will need to change the `expand_dist` parameter passed to the RRT planner from the `main.py` script.
5. For the simulator tasks, as long as the planners themselves work, we're happy. In fact, we expect there to be issues running in sim; we just want you to comment on them and think about why they occur.
6. For the controller, you'll probably only have to edit `controller.py`. Remember that the more time you spend on the controller theory the less time it'll take to tune and the better it'll end up working. At the same time, complicated controllers can be hard to properly implement and debug. Your implementation doesn't need to be novel or groundbreaking, it just needs to work well enough to demonstrate that your paths are executable. We're leaving this part deliberately open-ended. You'll have to figure out a control law and prove to yourself that it works, you'll have to write all the code to implement it and plot the results, and you'll have to tune it until you're satisfied. Note that you only have to get good tracking in sim. When you run on the robot, it'll likely work much less well. We just want to know why.

4.7 Using the TurtleBots

A couple of reminders when using the TurtleBots:

- Remember to switch them ON to charge. If a TurtleBot is not sufficiently charged for the next group to use you may lose points.
- Carry them from underneath the base, not the acrylic platforms.
- Watch where they're going! The TurtleBots don't have an E-Stop button so the fastest way to stop them is to press Ctrl + C, and you should be ready to do so whenever running your plans on the robot. There isn't a lot of space in the lab, so be especially careful so the TurtleBots don't crash into the furniture or your classmates.

- You can refer to the Robot Usage Guide for the process of setting up the TurtleBot. Remember to follow all of the rules in the Robot Usage Guide.

After setting up the TurtleBot, ssh into the TurtleBot and run

```
rosrun tf static_transform_publisher -1 -1 0 0 0 0 odom fake_odom 1
```

This is a hack to get the TurtleBot to think its starting state is at $(1, 1, 0, 0)$. To run the plans on the TurtleBot, you can use the same launch file and run

```
roslaunch proj2_pkg init_env.launch map:=empty.yaml start_position:="1 1 0" sim:=false
```

This will prepare the environment to run the plans on the actual TurtleBot when you run the `main.py` script.

4.8 Common Issues

- If the simulator does not seem to be following your paths, it is possible that your plan is causing the robot to hit its steering limit and it is unable to recover afterwards. You should double check and potentially make a more conservative planner with respect to the maximum steering angles (and maybe also with the input limits); however, there will always be some drift and an open loop control can not correct itself.
- In your sinusoids implementation, be careful not to hard-code any references to time. You'll want to play with how much time you give each subpath in order to compensate for input constraints.
- Don't be worried if the optimization planner takes a while to run.

5 Scoring

Table 1: Point Allocation for Project 2

Section	**Points**
Page Limit	5
Figures: Quality and Readability	5
Abstract	5
Methods: Sinusoids	10
Methods: Optimization	5
Methods: RRT	10
Results	10
Discussion	10
Discussion (Controller)	10*
Bibliography	4
Code	3
Video	3
Bonus:	5*

Summing all this up, for undergrads this project will be out of 70 points, with an additional 15 points possible. For grad students, the project will be out of 80 points, with an additional 5 points possible.

6 Submission

You'll submit your writeup on Gradescope and your code to GitHub Classroom. Your code will be checked for plagiarism, so please submit your own work. Only one submission is needed per group. Please add your groupmates as collaborators on both GitHub and Gradescope.

7 Improvements

If you notice any typos or things in this document or the starter code which you think we should change, please let us know. Next year's students will thank you.

8 References

[1] R. M. Murray, S. S. Sastry, and L. Zexiang. *A Mathematical Introduction to Robotic Manipulation*. 1st. USA: CRC Press, Inc., 1994.

[2] R. M. Murray and S. S. Sastry. "Nonholonomic motion planning: steering using sinusoids". *IEEE Transactions on Automatic Control* 38.5 (May 1993).

[3] M. Zucker et al. "Chomp: Covariant hamiltonian optimization for motion planning". *The International Journal of Robotics Research* 32.9-10 (2013).

[4] J. Schulman et al. "Motion planning with sequential convex optimization and convex collision checking". *The International Journal of Robotics Research* 33.9 (2014).

[5] M. Kelly. "An introduction to trajectory optimization: How to do your own direct collocation". *SIAM Review* 59.4 (2017).

[6] S. Bansal et al. "Combining Optimal Control and Learning for Visual Navigation in Novel Environments". *CoRR* abs/1903.02531 (2019). URL: http://arxiv.org/abs/1903.02531.

[7] R. Walambe et al. "Optimal Trajectory Generation for Car-type Mobile Robot using Spline Interpolation." *IFAC-PapersOnLine* 49.1 (2016). 4th IFAC Conference on Advances in Control and Optimization of Dynamical Systems ACODS 2016. URL: http://www.sciencedirect.com/science/article/pii/S2405896316301215.

[8] N. D. Ratliff, J. Issac, and D. Kappler. "Riemannian Motion Policies". *CoRR* abs/1801.02854 (2018). URL: http://arxiv.org/abs/1801.02854.

[9] X. Meng et al. "Neural Autonomous Navigation with Riemannian Motion Policy". *CoRR* abs/1904.01762 (2019). URL: http://arxiv.org/abs/1904.01762.

[10] F. Borrelli, A. Bemporad, and M. Morari. *Predictive Control for Linear and Hybrid Systems*. 1st. USA: Cambridge University Press, 2017.

[11] URL: https://www.cs.tufts.edu/comp/150IR/hw/cspace.html.

[12] S. M. Lavalle. *Rapidly-Exploring Random Trees: A New Tool for Path Planning*. Tech. rep. 1998.

[13] S. M. LaValle. *Planning algorithms*. Cambridge university press, 2006.

[14] L. E. Dubins. "On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents". *American Journal of mathematics* 79.3 (1957).

[15] J. Reeds and L. Shepp. "Optimal paths for a car that goes both forwards and backwards". *Pacific journal of mathematics* 145.2 (1990).

[16] B. Paden et al. *A Survey of Motion Planning and Control Techniques for Self-driving Urban Vehicles*. 2016.

[17] X. Xu et al. "Realizing simultaneous lane keeping and adaptive speed regulation on accessible mobile robot testbeds". *2017 IEEE Conference on Control Technology and Applications (CCTA)*. Aug. 2017.

[18] J. Kong et al. "Kinematic and dynamic vehicle models for autonomous driving control design". *2015 IEEE Intelligent Vehicles Symposium (IV)*. June 2015.

[19] H. Fan et al. *Baidu Apollo EM Motion Planner*. 2018.

[20] R. Carona, A. P. Aguiar, and J. Gaspar. "CONTROL OF UNICYCLE TYPE ROBOTS Tracking, Path Following and Point Stabilization". Nov. 2008.

[21] G. Walsh et al. "Stabilization of trajectories for systems with nonholonomic constraints". *IEEE Transactions on Automatic Control* 39.1 (Jan. 1994).